Tree Predictors for binary classification

DSE - Machine Learning
Melany Y. Gomez Herrada (39351A)
melanyyohana.gomezherrada@studenti.unimi.it

October 2024

1 Introduction

The goal of this project was to develop a tree-based predictor from scratch for binary classification. The dataset used contains detailed characteristics of mushrooms, which are used to classify them as either edible or poisonous. The core task is to determine a mushroom's edibility based on its attributes. To achieve this, a decision tree was constructed, where each internal node applies a binary test on a single feature to guide the classification process.

The first part of project involved constructing a class for the nodes of the tree, which included initializing nodes, managing left and right child nodes, and distinguishing between internal nodes and leaf nodes. To it followed the construction of a full binary tree classifier, using splitting criteria and stopping criteria to control the tree's growth and performance.

An important component of this project was the tuning of hyperparameters related to both the splitting and stopping criteria. For splitting, metrics such as the Gini index, entropy and miss-classification error were used, while for stopping the criteria introduced were: a maximum depth limit and a maximum number of nodes. Finally the model's performance was evaluated based on the training error using 0-1 loss.

This report details the implementation process, discusses the methodology used, and provides a critical analysis of the findings, including recommendations for improving model performance.

2 Data Cleaning and Management

2.1 About the Dataset

This project uses a dataset containing 61,069 mushroom samples with 21 columns that detail their physical features and edibility. The main target variable, class, indicates whether each mushroom is edible or poisonous.

The dataset includes both numerical and categorical features. For example, cap-diameter, stem-height, and stem-width are numerical measurements that describe the size of the mushrooms and features like cap-shape, cap-color, and gill-color are categorical, describing visible characteristics that help distinguish different types of mushrooms.

2.2 Data cleaning

The initial stage of our data analysis involved a data cleaning and management process to ensure the quality and usability of the Mushroom Dataset. Below is a summary of the steps followed to transform and prepare the dataset for further analysis.

The dataset was loaded from a CSV file using Pandas, specifying the correct delimiter (';'). The Initial exploration consisted in displaying the first few rows `'df.head()'`, generating

summary statistics 'df.describe()', and checking the data structure 'df.info()'. These steps helped in understanding the data's format, value ranges, and identifying potential issues like missing values.

To improve the dataset quality, the following actions were taken to **manage missing values**:

1. Removing Columns with High Missing Values: Columns with more than 30% missing values were eliminated from the dataset. The proportion of missing data was calculated using 'df.isnull().mean() * 100', and columns exceeding the threshold were identified and dropped. This ensured that the majority of features retained enough data to be meaningful for analysis. The removed columns were noted for transparency.

Columns	Type	Missing Values (%)
gill-spacing	object	41.04
stem-root	object	84.39
stem-surface	object	62.43
veil-type	object	94.80
veil-color	object	87.86
spore-print-color	object	89.60

Table 1: Columns dropped, >30% of missing values

1. Removing Rows with Missing Values: After reducing the dataset by dropping columns, rows with any remaining missing values were removed using 'df.dropna()'. This resulted in a clean dataset without any missing values, providing a robust base for further analysis.

Number of rows	Number of columns	Total data points
37065	15	555975

Table 2: Dataset size post cleaning

The target variable, 'class', which represents the type of mushroom (edible or poisonous), was converted to a numerical representation. Specifically, the values 'e' (edible) and 'p' (poisonous) were mapped to 0 and 1, respectively. This conversion was not strictly necessary, I could have used 'e' and 'p' but converting them to numerical values helps to maintain consistency, reduces the likelihood of compatibility issues, and simplifies the handling of calculations in custom implementations. It is often a recommended to convert categorical target variables to numerical labels to avoid unexpected issues.

The feature columns were identified, excluding the target variable. The features ('X') and labels ('Y') were then prepared:

- 'X' contained the original categorical feature values, while 'Y' contained the numerical labels.Tree predictors are suitable for structured datasets that involve both categorical and numerical features.

- Both were extracted as NumPy arrays, with 'Y' explicitly converted to an integer type for consistency.

To address potential class imbalance, class frequencies were computed using NumPy's 'np.unique()' function. The frequencies were used to derive class weights, assigning lower weights to more frequent class and higher weights to less frequent one, helping to mitigate any bias in the dataset. Class weights were computed as '1.0 - frequency' for each class, ensuring

that less represented classes received a proportionately higher weight. These weights were then used to create a sample weight array, which could later be used to emphasize underrepresented classes during model training. Actually the distribution between the two classes is quite balanced so calculating class weights to emphasize underrepresented classes is not strictly necessary, we could also proceed without applying class weight or test both options to see which one gets better results.

0	**1**
0.46	0.54

Table 3: Class Frequencies

To prepare the data for model training and evaluation, **the dataset was split into training and test sets**:

- The dataset was shuffled using a fixed seed `‘np.random.seed(42)‘` for reproducibility.
- An 80/20 split ratio was applied to allocate 80% of the data for training and 20% for testing.

The resulting training and testing sets (`X_train`, `X_test`, `Y_train`, `Y_test`) included corresponding sample weights (`sample_weight_train`, `sample_weight_test`) to apply during model training.

These were the steps followed for data cleaning and management , ensuring that the dataset was properly prepared for the following analysis, handling missing data, ensuring balanced representation of classes, and providing clean, usable data for model training and evaluation.

3 Decision Tree Classifier Model Development

Decision trees are a versatile type of supervised machine learning algorithm used for classification and regression tasks, particularly well-suited for handling data with heterogeneous feature types. A decision tree predictor is structured as an ordered, rooted tree where each internal node represents a test or condition on a feature, each branch corresponds to the outcome of that test, and each leaf node represents a final decision or class label. The path from the root node to a leaf node forms a rule that can be used to make predictions on new data.

At each node in the tree, the algorithm selects the feature and corresponding threshold that best splits the data into homogeneous subsets concerning the target variable. This selection is based on a splitting criterion, such as the Gini impurity, entropy, or misclassification error, which measures the impurity or disorder in the dataset. The goal is to minimize this impurity by creating nodes that are as pure as possible in terms of class distribution.

The tree construction process is recursive and continues until a stopping condition is met. Stopping criteria can include reaching a maximum tree depth, having a minimum number of samples required to split a node, or achieving a minimum impurity decrease. By adjusting these parameters, one can control the complexity of the tree and address issues like overfitting.

Advantages and limitations of Decision Trees Decision trees naturally manage different types of data (numerical, ordinal, categorical) without the need for encoding or scaling, making them suitable for datasets with features that vary across incomparable domains. The tree structure provides an explicit set of rules that can be easily visualized and interpreted. They are non-parametric models that do not assume any underlying distribution of the data. They are flexible in capturing complex patterns and interactions between features. During the tree-building process, trees perform implicit feature selection by choosing the most informative

features for splitting. This can reduce dimensionality and highlight the most relevant variables influencing the target outcome.

Decision trees can easily overfit the training data, especially when they become too deep and capture noise instead of the underlying pattern. This results in poor generalization to unseen data. Controlling the tree's growth through appropriate stopping criteria and pruning techniques is essential to prevent overfitting. Trees may become biased when dealing with features that have many levels or when certain classes dominate the dataset. Features with more unique values can dominate the splitting process, not necessarily because they are more informative but because they offer more opportunities to reduce impurity. Small changes in the training data can lead to different splits and, consequently, a different tree structure. This high variance makes individual decision trees less robust compared to ensemble methods like Random Forests.

3.1 Function build_Tree()

Building a decision tree from a training dataset involves a recursive process that begins with initializing a single-node tree and continues until a stopping criterion is met. The process for the implementation of the algorithm is outlined below:

Initialization The decision tree construction starts with a single-node tree where all training data is associated with the **root node**, the basis of the tree. Initially, this root node represents the entire dataset, it has not yet been split. The root is assigned the class label that is most frequent in the entire training set. The function will begin partitioning this initial node based on the "best splitting feature", the feature characterized by the maximum reduction in impurity.

Stopping Criteria After the initial split the function immediately checks if it should stop splitting. This model follows two stopping criteria:

- If `len(unique_classes)` == 1:, if all data points within a node belong to the same class, there is no need to split further, and the node is converted to a leaf node with the label assigned to that class.

- If the number of samples is less than the `min_samples_split` threshold, or if the current depth has reached the `max_depth`, the node becomes a leaf node, and a class label is assigned based on `calculate_leaf_value()`.

These criteria are designed to control the complexity of the tree and help prevent overfitting by ensuring that the model does not become overly tailored to the training data.

- `min_samples_split`:

  This parameter defines the minimum number of samples required to split an internal node. Setting a higher value for `min_samples_split`, prevents the model from creating nodes that are too specific, which can lead to overfitting. For instance, if set to 10, a node will not split unless it contains at least 10 samples. This parameter effectively controls the tree's growth a tree with too many splits can capture noise rather than the underlying data pattern, making it less effective on unseen data.

- `max_depth`:

  This parameter specifies the maximum depth of the tree. A deeper tree can model more intricate relationships in the training data but may fail to generalize well to new data. For example, if `max_depth` is set to 3, the tree will only have 3 levels of decisions, which simplifies the model, keeping the tree shallow, focuses on broader patterns instead of getting lost in the intricacies of the training data.

Evaluating Splits If all samples do not belong to a single class or if stopping criteria are not met the `get_best_split()` function is used. In decision trees, finding the best split at each internal node is a critical part of the tree-building process. For each feature, the algorithm checks various points at which the dataset can be split.

In decision trees, differentiating between numerical and categorical features is crucial because the algorithms for splitting these features vary significantly. By accurately detecting the feature types, the algorithm can apply the appropriate split criteria, thereby optimizing the tree's decision-making capability and ensuring effective learning from the data.

- For numerical features, potential split points are typically evaluated by checking thresholds between unique values.
- For categorical features, splits are evaluated by testing groupings of different categories.

By accurately detecting the feature types, the algorithm can apply the appropriate split criteria, thereby optimizing the tree's decision-making capability and ensuring effective learning from the data.

The goal is to choose the split that results in the most significant decrease in impurity, thereby making the subsets of data more homogeneous in terms of the target variable. In this model the criteria used to evaluate splits are:

- **Gini Index:**
$$\psi(p) = 2p(1-p),$$
where p is the proportion of positive instances. It measures the impurity of a node by assessing the probability of misclassifying a randomly chosen sample
- **Entropy:**
$$\psi(p) = -\frac{p}{2}\log_2 p - \frac{1-p}{2}\log_2(1-p),$$
measures the disorder or uncertainty in the data. A pure node has low entropy
- **Misclassification Error:**
$$\psi(p) = \min\{p, 1-p\}.$$
the simplest criterion, representing the fraction of misclassified samples.

These functions guide the selection of the best split by measuring the homogeneity of the resulting child nodes. Gini impurity and entropy are preferred over simple misclassification error because they are more sensitive to changes in class proportions, particularly when dealing with imbalanced datasets (not our case).

Splitting Nodes After choosing a splitting criterion, the algorithm evaluates the information gain or reduction in impurity by comparing the parent node's impurity with the weighted sum of the impurities in the child nodes. The information gain formula is:

$$\text{Information Gain} = H(\text{Parent}) - \left(\frac{N_{\text{Left}}}{N} \times H(\text{Left}) + \frac{N_{\text{Right}}}{N} \times H(\text{Right})\right)$$

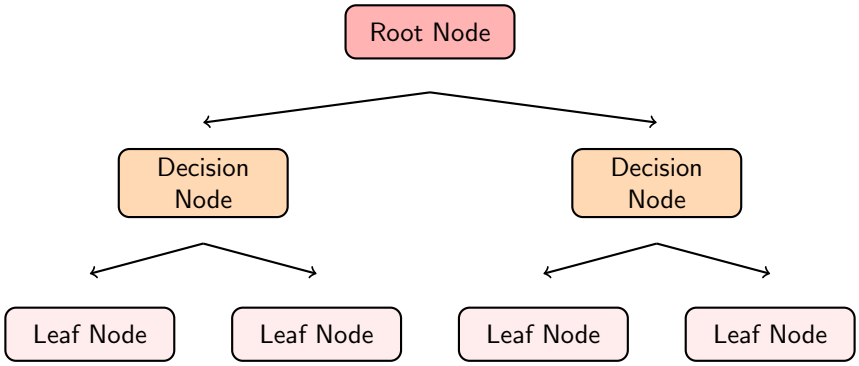
The split that results in the highest information gain is chosen as the best split for that node.

If a valid split is found `(best_split["info_gain"]⩾ self.min_impurity_decrease)` ,the function recursively generates two child nodes, referred to as `left` and `right` children:

- The left child node `(left_child)` is constructed using the data points that meet the split criterion `(best_split["dataset_left"])`.

- The right child node (`right_child`) is constructed using the remaining data points (`best_split["dataset_right"]`).

The `build_tree()` function continues this recursive process for both branches until the stopping criteria are met at the child nodes, resulting in a combination of internal and leaf nodes that define the decision tree structure.



In this model if the node is an internal decision node, the function returns a dictionary representing the internal node with several attributes:

- **'feature'**: The feature index used for splitting.
- **'threshold'**: The threshold or category value used for the split.
- **'left' and 'right'**: The left and right child nodes, created recursively.
- **'info_gain'**: The information gain from the split.
- **'is_leaf'**: Set to False, indicating that this is not a leaf node.

4 Model Training and Evaluation

4.1 Hyperparameter Tuning

Hyperparameter tuning is a critical step in building effective machine learning models, as it involves selecting the set of hyperparameters that yields the best performance on unseen data. Hyperparameters are configuration settings used to control the learning process and the structure of the model. Unlike parameters learned during training, they are not directly derived from the data. In the context of decision trees, hyperparameters significantly influence the model's ability to generalize, control overfitting, and computational efficiency.

This model focused on tuning the following hyperparameters:

- **Splitting Criteria**: the choice of splitting criterion affects how the tree partitions the data at each node. Gini Index and Entropy are more sensitive to node purity and often result in better performance than Misclassification Error.

- **Maximum Tree Depth**: depths of 10 and 20 were tested to control the complexity of the tree. A deeper tree can capture more complex patterns but may overfit by capturing noise. By setting a depth limit, the risk of overfitting is reduced and the model's ability to generalize to unseen data is improved.
- **Minimum Samples per Split**: it specifies the minimum number of samples required to split an internal node, it ensures that each split is statistically significant and contributes meaningfully to the model's predictive power. In the model, considered values of 2 and 5 to determine the smallest node size eligible for splitting, a higher value prevents the tree from creating nodes with few samples, which can be unreliable.

4.2 Cross-Validation

To evaluate the performance of different hyperparameter combinations, a 5-fold cross-validation method was employed. Cross-validation is a resampling procedure used to assess how the results of a statistical analysis will generalize to an independent dataset. It is widely used to prevent overfitting and to estimate the model's predictive performance.

1. **Data Partitioning**:
   - The training dataset was randomly partitioned into five equal-sized subsets (folds).
   - In each of the five iterations, one fold was held out as the validation set, and the remaining four folds were used for training.
   - This process ensures that every sample in the dataset is used exactly once for validation and four times for training.
2. **Model Training and Evaluation**:
   - For each hyperparameter combination, the model was trained on the training folds and evaluated on the validation fold.
   - The performance metric, in this case, accuracy (or equivalently, error rate), was recorded for each fold.
   - The process was repeated five times, with each fold serving as the validation set once.
3. **Performance Metrics**:
   - The **average cross-validation accuracy** was computed by averaging the accuracy scores across all five folds:

$$\text{Average Accuracy} = \frac{1}{k} \sum_{i=1}^{k} \text{Accuracy}_i,$$

     where $k = 5$ is the number of folds, and Accuracy_i is the accuracy on the i-th validation fold. This average provides a robust estimate of the model's generalization performance on unseen data.

4.3 Hyperparameter Grid Search

Hyperparameter Grid Search is a technique used to automatically explore and find the optimal combination of hyperparameters that yields the best model performance. Unlike parameters (which the model learns during training), hyperparameters are predefined values that control the behavior and performance of the learning process.

The grid search was conducted over all possible combinations of the following hyperparameters:

- **Splitting Criteria**: {Gini, Entropy, Misclassification Error}
- **Maximum Tree Depth**: {10, 20}
- **Minimum Samples per Split**: {2, 5}

This resulted in $3 \times 2 \times 2 = 12$ combinations. For each combination, a new instance of the decision tree classifier was created with the current set of hyperparameters. The model was evaluated using 5-fold cross-validation and average cross-validation accuracy was calculated. The hyperparameters and corresponding average accuracies were recorded for comparison.

- **Fixing the depth to 10 and 20** allows the model to explore more complex relationships within the data, which can be beneficial for datasets with intricate patterns. This deeper tree can improve model performance but it also carries a higher risk of overfitting.
- **Setting the miminum samples per split to 2 and 5** allows for the tree to grow and split even with a small number of samples. This can be useful in capturing more detailed patterns but it can also increase the risk of overfitting, careful validation is needed to ensure it doesn't result in poor generalization.

4.4 Implementation Details

To efficiently perform hyperparameter tuning on this large dataset, we implemented several strategies aimed at enhancing computational efficiency, increasing processing speed, and ensuring the reliability of the results.

- **Parallel Processing**:
  - The hyperparameter tuning process involved evaluating multiple combinations of hyperparameters, which can be computationally intensive. The evaluations were parallelized using the `multiprocessing` library. This allowed simultaneous computation across multiple CPU cores, significantly reducing the total computation time.
- **Reproducibility**:
  - A fixed random seed was used to initialize the random number generator, this seed was applied wherever randomness was involved, ensuring that the data partitioning and cross-validation splits were consistent across runs. Reproducibility makes it easier to verify results and identify issues.

4.5 Results

The hyperparameter tuning process revealed the following results:

Best Hyperparameter Combination:

- **Splitting Criterion**: Entropy
- **Maximum Tree Depth**: 20
- **Minimum Samples per Split**: 2

An **average cross-validation accuracy** of **99.70%** was obtained. The high accuracy indicates that the model generalizes well to unseen data.

Interpretation of the results:

Crit.	Max Depth	Min Split	Avg CV Err.	Avg CV Acc.
gini	10	2	0.0773	0.9227
gini	10	5	0.0775	0.9225
gini	20	2	0.0032	0.9968
gini	20	5	0.0039	0.9961
entropy	10	2	0.0992	0.9008
entropy	10	5	0.0998	0.9002
entropy	**20**	**2**	**0.0030**	**0.9970**
entropy	20	5	0.0047	0.9953
error	10	2	0.1322	0.8678
error	10	5	0.1323	0.8677
error	20	2	0.0664	0.9336
error	20	5	0.0673	0.9327

Table 4: Hyperparameter Tuning Results

- **Entropy as Splitting Criterion**:
  - Entropy tends to produce more balanced trees and is sensitive to changes in class probabilities.
  - It is particularly effective in datasets where the classes are not perfectly separable using simple splits.
- **Maximum Tree Depth of 20**:
  - Allows the model to capture complex patterns in the data.
  - The depth of 20 is sufficient to model interactions between features without overfitting, as indicated by the cross-validation results, it balances trade-off between bias and variance
- **Minimum Samples per Split of 2**:
  - Allowing a minimum of 2 samples per split provides the decision tree with maximum flexibility to partition the data. This setting enables the model to create splits even when only a few samples meet the criteria, which can be crucial for capturing rare but significant patterns, on the other hand the risk of modeling noise increases.
  - Cross-validation helped ensure that the model generalizes well.

5 Model Results

5.1 Classification Accuracy and Error Rates

Training Error and Accuracy The model achieved a training error of 0.0004, indicating that only 0.04% of the training samples were misclassified.

$$\begin{aligned}\text{Training Accuracy} &= 1 - \text{Training Error} = 1 - 0.0004 = 0.9996 \\ &= 99.96\%\end{aligned}$$

Test Error and Accuracy On the test set, the error was 0.0032, showing a misclassification rate of 0.32% on unseen data.

$$\begin{aligned}\text{Test Accuracy} &= 1 - \text{Test Error} = 1 - 0.0032 = 0.9968 \\ &= 99.68\%\end{aligned}$$

These high accuracy rates demonstrate that the model not only fits the training data exceptionally well but also generalizes effectively to new, unseen data.

5.2 Confusion Matrices

To gain deeper insights into the model's performance, especially concerning the types of errors made, confusion matrices for both the training and test sets were computed.

Note: In binary classification, the classes are often labeled as positive (e.g., poisonous mushrooms) and negative (e.g., edible mushrooms).

Table 5: Confusion Matrix for the Training Set

		Predicted		
		Negative	Positive	Total
Actual	Negative	13,483	3	13,486
	Positive	8	16,158	16,166
	Total	13,491	16,161	29,652

Training Set Confusion Matrix

- **True Negatives (TN)**: 13,483 correctly predicted edible mushrooms.
- **False Positives (FP)**: 3 edible mushrooms incorrectly predicted as poisonous.
- **False Negatives (FN)**: 8 poisonous mushrooms incorrectly predicted as edible.
- **True Positives (TP)**: 16,158 correctly predicted poisonous mushrooms.

Table 6: Confusion Matrix for the Test Set

		Predicted		
		Negative	Positive	Total
Actual	Negative	3,446	12	3,458
	Positive	12	3,943	3,955
	Total	3,458	3,955	7,413

Test Set Confusion Matrix

- **True Negatives (TN)**: 3,446 correctly predicted edible mushrooms.
- **False Positives (FP)**: 12 edible mushrooms incorrectly predicted as poisonous.
- **False Negatives (FN)**: 12 poisonous mushrooms incorrectly predicted as edible.
- **True Positives (TP)**: 3,943 correctly predicted poisonous mushrooms.

5.3 Interpretation of Results

The confusion matrices reveal that the model has:

- **Extremely Low Misclassification Rates**: Both the false positive and false negative counts are minimal in the training and test sets.
- **Balanced Performance**: The model performs consistently across both classes, which is crucial in applications where misclassification can have serious consequences.
- **High Sensitivity and Specificity**:
  - **Sensitivity**: measures the proportion of actual positive cases (poisonous mushrooms) that are correctly identified by the model.

$$\text{Sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

    * **Training Set**:

$$\frac{16,158}{16,158 + 8} = \frac{16,158}{16,166} \approx 0.9995 \text{ or } 99.95\%$$

    * **Test Set**:

$$\frac{3,943}{3,943 + 12} = \frac{3,943}{3,955} \approx 0.9970 \text{ or } 99.70\%$$

  - **Specificity** : also known as true negative rate, measures the proportion of actual negative cases (edible mushrooms) that are correctly identified by the model.

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

    * **Training Set**:

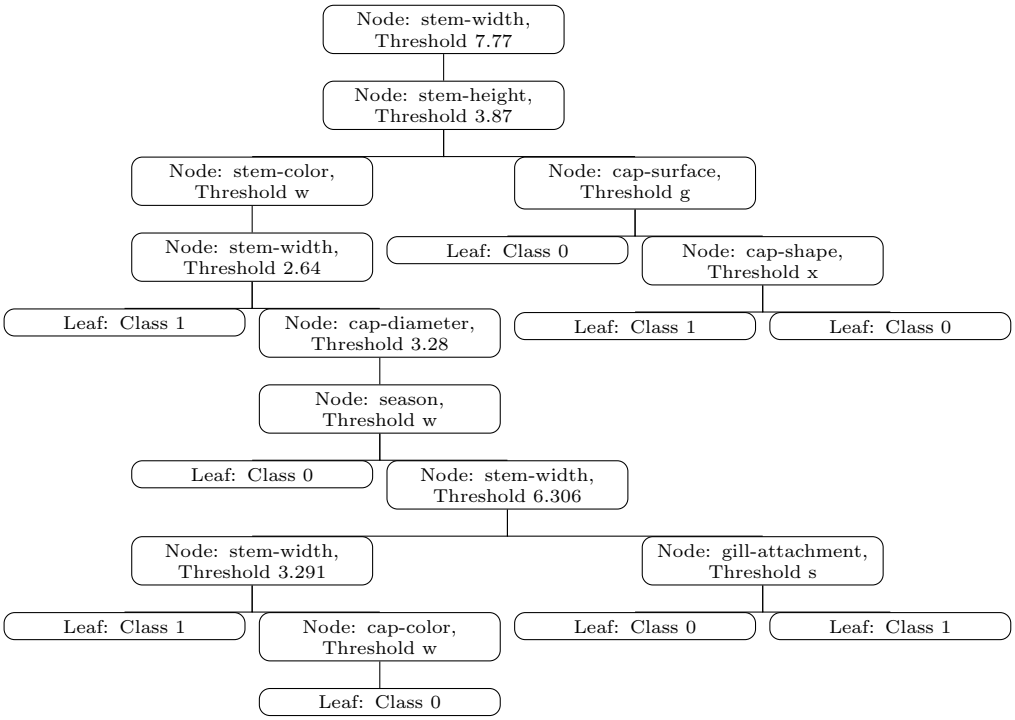
$$\frac{13,483}{13,483 + 3} = \frac{13,483}{13,486} \approx 0.9998 \text{ or } 99.98\%$$

    * **Test Set**:

$$\frac{3,446}{3,446 + 12} = \frac{3,446}{3,458} \approx 0.9965 \text{ or } 99.65\%$$

These metrics indicate that the model is highly effective at correctly identifying both poisonous and edible mushrooms.

5.4 Tree visualization



The scheme presented visually illustrates the decision tree structure. It provides a clear, hierarchical view of how the tree makes decisions based on the features and their thresholds. The root node at the top represents the initial feature, stem-width, with a threshold value of 7.77. The tree then branches into subsequent nodes, where each internal node indicates a decision point based on different features (e.g., stem-height, stem-color, cap-surface) and their respective thresholds.

Each leaf node signifies a final classification outcome. Reaching a leaf labeled Class 1 indicates the mushroom is classified as poisonous, while a leaf labeled Class 0 indicates it is edible. This structured approach allows for clear interpretation of the decision-making process of the model, showcasing how specific features influence the classification of mushrooms.

6 Conclusion

In this project, we successfully implemented a decision tree classifier from scratch to classify mushrooms as either edible or poisonous based on their attributes. The process involved meticulous data cleaning, hyperparameter tuning, and model evaluation, which culminated in a highly effective classification model.

The data cleaning phase ensured that our dataset was free of missing values, leading to a robust basis for the following analysis. Converting the categorical target variables into numerical labels and addressing potential class imbalance through the calculation of class weights, was a plus for this analysis but had the aim of enhancing the dataset's usability and ensuring that the model would be trained on a representative sample.

The model development took advantage of the important characteristics of decision trees, such as their capability to work with both numerical and categorical data types, as well as their inherent interpretability, which allows for easy visualization and understanding of the decision-making process. Hyperparameter tuning was a crucial step in optimizing the model's performance, with the best configuration identified as using the entropy criterion, a maximum tree depth of 20, and a minimum samples per split of 2. This configuration achieved an impressive average cross-validation accuracy of 99.70%, indicating that the model not only learned effectively from the training data but also generalizes well to unseen examples.

The model demonstrated extremely low misclassification rates, as evidenced by the confusion matrices for both training and test sets, highlighting its balanced performance across classes. With a training accuracy of 99.96% and a test accuracy of 99.68%, the decision tree classifier effectively captured the complexities of the data.

Future work could explore different hyperparameters, the integration of ensemble methods, such as Random Forests, to further improve classification accuracy and robustness. Overall, the project successfully met its objectives and showcases the effectiveness of decision tree classifiers in binary classification tasks.

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.